

Enregistrer ses objets dans des fichiers

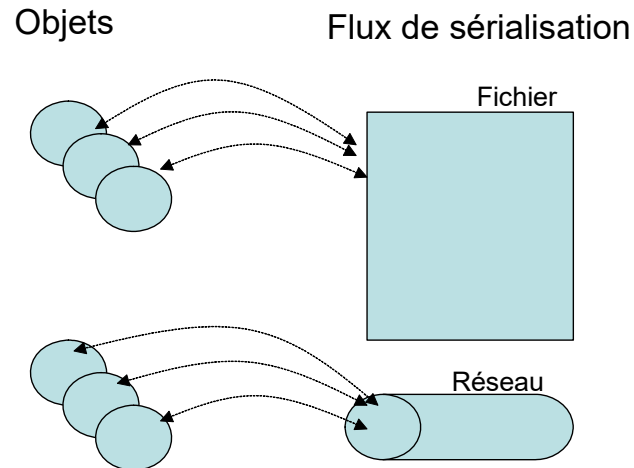


serialisation

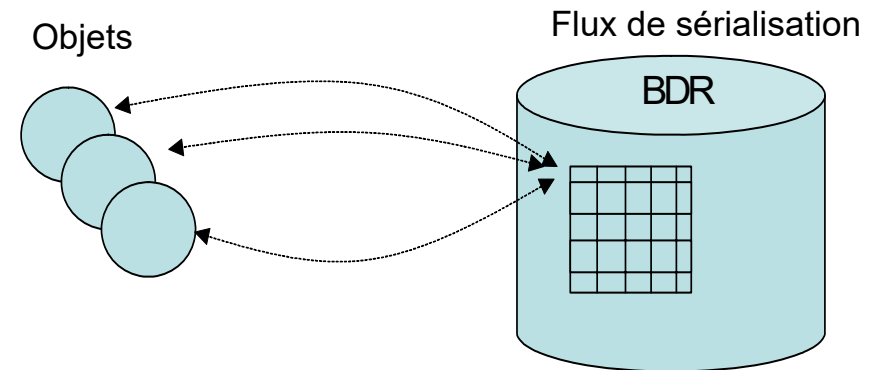
- La durée de vie d'un objet peut devoir être supérieure au temps d'exécution de l'application qui le crée et l'utilise.
- Dans ce cas, il n'est plus lié à une application et peut être partagé par des applications s'exécutant à des instants différents.
- Un objet qui conserve son état (i.e. la valeur de chacune de ses données membres) quand il est libéré en mémoire puis chargé par une application possède la "**propriété de persistance**".
- **Persistance** : capacité d'une application à enregistrer les instances.
- **Exemple** : un compte bancaire doit persister avec ses nouvelles données après son utilisation pour une opération de transfert d'argents.
- Il faut sauvegarder l'objet dans un fichier (que ce soit un fichier utilisateur ou via un SGBD).
- La question du SGBD relationnel ou objet pourrait se poser si la prédominance des SGBDR n'imposait pas ce type d'organisation des données.

- Le besoin de persistance est un besoin :
 - **très récurrent** aux applications dès qu'elles manipulent un temps soit peu des données,
 - assez **régulier** dans ses **fonctionnalités** : enregistrer l'objet, charger l'objet, ...,
 - fastidieux à programmer (penser aux objets composés d'autres objets),
 - sans VA spectaculaire (c'est un attendu implicite du client).
- On a cherché à automatiser cette tâche. Tous les langages évolués proposent des solutions pour **gérer** la persistance des objets
 - Solutions souvent adossées à des BD (solutions intégrées dans les serveurs d'application actuels : Websphere - IBM, WebLogic - BEA, Oracle, .Net - MS, ...).
- En Java, le concept de persistance est fortement lié à celui de sérialisation d'objet.

- C'est un mécanisme Java pour générer un flux à partir d'un objet : le mettre en "série d'octets".
- Ce mécanisme est aussi bien valable pour un flux de type *File* qu'une *Socket* TCP/IP :



- Le cas de la sauvegarde dans une BD est un peu plus complexe : implique de faire un *mapping* entre des objets et leurs représentation en relationnel.



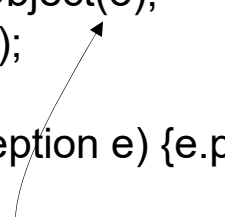
- La sérialisation permet d'enregistrer des arborescences d'objets interconnectés et de recharger ces objets ultérieurement.
- Pour bénéficier de ce mécanisme, un objet doit implémenter l'interface *java.io.Serializable*, sinon on lève l'exception *java.io.NotSerializableException*.
- Il faut noter que plusieurs classes de l'API Java implémentent déjà l'interface *Serializable* : *String*, *Integer*, ...

- L'interface `Serializable` s'utilise de façon particulière :
 - elle ne définit aucun champ ni aucune méthode : « interface de marquage » ,
 - Les instances de cette classe seront alors manipulables par les méthodes des classes `java.io.ObjectOutputStream` et `java.io.ObjectInputStream` :
 - `writeObject` pour la **sérialisation** ;
 - `readObject` pour la **désérialisation**,
 - ces 2 méthodes peuvent être redéfinies, si nécessaire : le comportement par défaut ne convient pas ;

Préparation des flux nécessaires

```
import java.io.*;

public void faitPersister(Object o)
{
    try
    {
        ObjectOutputStream buf1=null;
        buf1 = new ObjectOutputStream(new FileOutputStream("ObjetsInFile"));
        buf1.writeObject(o);
        buf1.close();
    }
    catch (Exception e) {e.printStackTrace();}
}
```



C'est à ce moment là qu'il y aura appel à la méthode *writeObject()* de la classe de *o*

Exemple

```
import java.io.Serializable;

public class Animal implements Serializable
{
    int age;
    boolean vaccin;
    String couleur;

    Animal() { this.age = 0;      this.vaccin = false;      this.couleur = "red"; }

    Animal(int a, boolean v, String c)
    {
        this.age = a;
        this.vaccin = v;
        this.couleur = c;
    }
}
```


Exemple : sauvegarde de l'objet

```
import java.io.Serializable;

public class Serialise
{
    public public static void main(String[] args)    throws IOException
    {
        Animal chien = new Animal(5, true, "noir");
        FileOutputStream fos = new FileOutputStream("SaveAnimal.pouet");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject(chien);
        oos.close();
    }
}
```

Exemple : chargement de l'objet

```
public class Deserialise
{
    public static void main(String[] args)
        throws IOException, ClassNotFoundException
    {
        FileInputStream fis = new FileInputStream(args[0]);
        ObjectInputStream ois = new ObjectInputStream(fis);

        Animal monObjetDeserialise = (Animal) ois.readObject();
        // manipulation classique de monObjetDeserialise
        ois.close();
    }

    ...
}
```

Le **serialVersionUID** est un "numéro de version", associé à toute classe implémentant l'interface *Serializable*
→ permet de s'assurer, lors de la désérialisation, que les versions des classes Java soient concordantes.
→ Si le test échoue, une *InvalidClassException* est levée.

Une classe sérialisable peut déclarer explicitement son serialVersionUID : attribut nommé "serialVersionUID" qui doit être **static**, **final** et de type **long**.

Exemple : chargement de l'objet

```
public class Animal implements Serializable
{
    int age;
    boolean vaccin;
    String couleur;
    private static final long serialVersionUID = 1212L;
    ...
}
```

Sinon affecté par défaut par JAVA (Java(TM) Object Serialization Specification), risque d'être différent entre deux compilations, d'un compilateur à un autre, d'une machine à une autre, ...

- Il est parfois nécessaire de modifier quelque peu le mécanisme de sérialisation par défaut, par exemple si l'on ne souhaite pas qu'un champ soit enregistré (dans le cas d'un champ spécifique à une session par exemple).
- On utilise un indicateur d'usage particulier pour indiquer qu'une donnée membre est "transitoire" : le mot-clé *transient*.

Exemple : attributs non persistants

```
import java.io.*;
import java.util.*;

public class MaClasse implements Serializable
{
    private String monPremierAttribut;
    private transient Date monSecondAttribut;
    private long monTroisiemeAttribut;

    // ... d'éventuelles méthodes ...
}
```

- Si les besoins sont plus spécifiques encore et ne peuvent être traités en n'utilisant que l'indicateur d'usage *transient*, on peut redéfinir les méthodes *readObject* et *writeObject* qui sont appelées pour le chargement et la sauvegarde des objets.

```
private void writeObject(java.io.ObjectOutputStream out)  
    throws IOException;  
  
private void readObject(java.io.ObjectInputStream in)  
    throws IOException, ClassNotFoundException;
```

Ces deux méthodes devront être redéfinies dans les classes dont les instances doivent passer par ces traitements particuliers

- **Autres méthodes accessibles ou redéfinissables dans la classe sérializable :**

- Dans un `writeObject()`, on peut invoquer une écriture par défaut :

`defaultWriteObject()`

Écrit les champs non-static et non-transient de la classe courante sur le flux. Appelable seulement depuis la méthode `writeObject` de la classe à sérialiser

- *`writeReplace()` : lorsqu'on veut que toute demande d'écriture écrive en fait un objet particulier*
- *`readResolve()` : lorsqu'on veut qu'une demande de lecture produise un objet particulier*

Exemple : sérialisation *personnalisée*, un exemple :

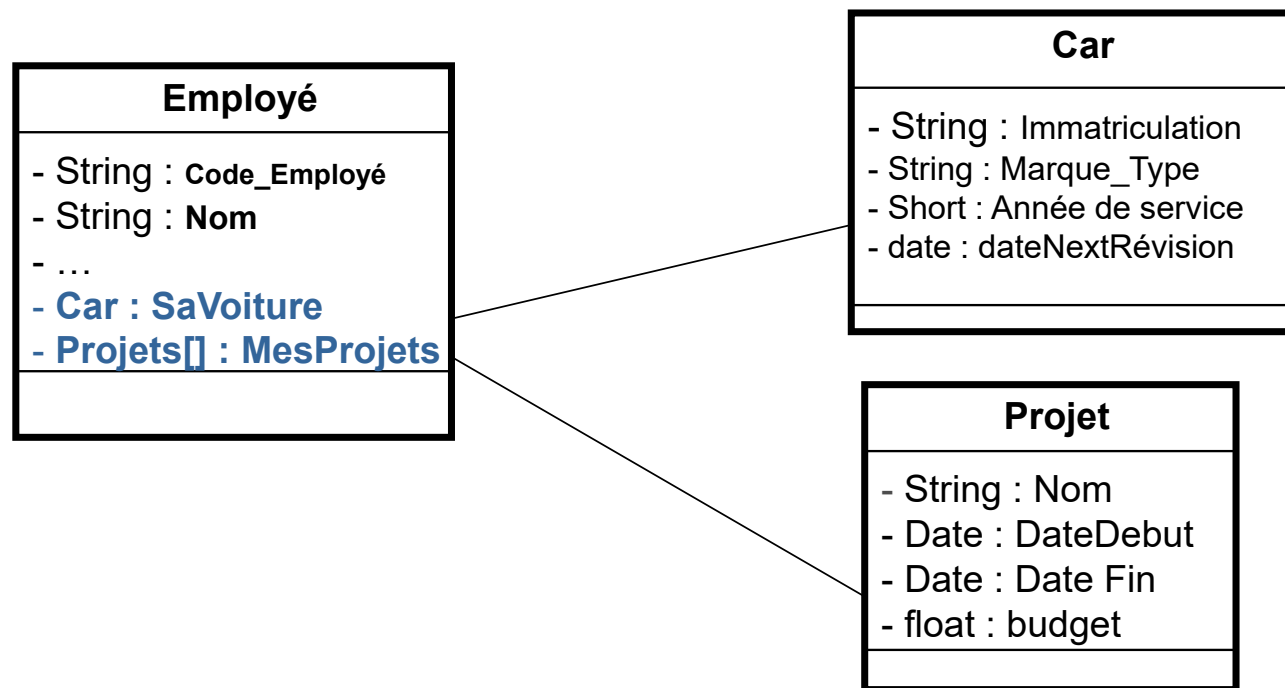
```
public class MaClasse implements Serializable
{
    private String Nom;
    private transient Date DateDuJour;
    private long Code;

    private void writeObject(ObjectOutputStream flux) throws IOException
    {
        flux.defaultWriteObject(); // Ecriture par défaut de OOS
        flux.writeChars("Quelque chose que je veux ajouter dans le fichier");
    }

    private void readObject(ObjectInputStream flux) throws IOException,
        ClassNotFoundException
    {
        flux.defaultReadObject(); // Lecture par défaut de OIS
        String s = flux.readLine(); // Récup de la chaîne enregistrée
        this.DateDuJour=// récupération de la date système;
    }
}
```

- Si une classe *sérialisable* étend une classe qui ne l'est pas :
 - les valeurs des données membres de l'objet parent ne seront pas écrites sur le flux de sortie *ObjectOutputStream*,
 - attention, lors de la reconstruction de l'objet, le constructeur **sans paramètre** sera appelé pour l'initialisation des données membres héritées de l'objet parent non sérialisable.
- Dans le cas d'une classe sérialisable étendant une classe elle-même sérialisable, la sérialisation se fait de façon classique (via celle de la classe et de sa super classe, ...).
- Les **sous-classes** d'une classe **sérialisable** sont elles-mêmes **sérialisables**.

- lorsqu'un objet est sauvé, tous les objets qui peuvent être atteints depuis cet objet sont également sauvés
- En particulier si l'on sauve le premier élément d'une liste, tous les éléments de la liste sont sauvés, ici aussi à condition bien sûr que tous ses éléments soient *sérialisables*).



- **Contexte :**

- les attributs d'un objet ne sont pas tous de type scalaire (*int, float, double, ...*), certains sont des agrégats (objets ou tableaux) : ce sont donc des références (pointeurs),
- Faut-il sérialiser la zone mémoire d'une référence qui, certes, contient les informations à un instant donné mais risque d'être libérée à tout moment (c'est le contenu de l'objet pointé qu'il faut sauvegarder) ?

- **Problèmes :**

- faut-il aussi déterminer si le type d'un attribut est un type scalaire ou un agrégat et adapter le comportement de la sérialisation ?
- comment sérialiser deux objets qui se pointent mutuellement (risque de boucle infinie, ce qui reviendrait à sauvegarder les mêmes objets indéfiniment) ?

Le flux de sortie *ObjectOutputStream* possède un certain nombre de mécanismes permettant de résoudre ces problèmes :

A chaque fois qu'un objet est sérialisé sur le flux, sa référence est rajoutée dans un vecteur (hashtable) trace de ces opérations.

Lorsqu'un nouvel objet demande son ajout au flux, JAVA vérifie préalablement dans le vecteur si sa référence existe ou non. Si elle existe, c'est qu'il a déjà été sérialisé

(cette seconde tentative d'écriture produit une référence vers la première copie de l'objet qui est en train d'être écrit),

- ➔ tout objet **n'est sauvé qu'une fois** grâce à un mécanisme de cache
- ➔ (il est donc possible de sauver une liste circulaire, à condition que tous ses éléments soient sérialisables),

- Ce mode de fonctionnement peut poser un problème, notamment si on doit sérialiser beaucoup d'objets.
- ⇒ risques que le vecteur associé aux flux de sérialisation « grossisse » jusqu'à ne plus tenir en mémoire (*java.lang.OutOfMemoryError*).
- On a la possibilité de pouvoir réinitialiser le flux (et donc le vecteur associé) pendant une opération de sérialisation grâce à la méthode *reset* du flux.
 - Solution pour la surcharge de mémoire **MAIS** cette solution ne peut être utilisée que si on est certain que les objets à sérialiser après l'appel de la méthode *reset* (sur le flux de sortie) n'ont pas déjà été sauvegardés dans le flux.
 - Si tel n'est pas le cas, certaines instances pourraient éventuellement être sérialisées 2 fois.

```
public class MaFenêtre
{
    // Créer une fenêtre et la sérialiser dans un fichier
    JFrame window = new JFrame("Ma fenêtre");
    JPanel pane = (JPanel)window.getContentPane();
    JTextArea textArea = new JTextArea("Ceci est le contenu !!");

    Public MaFenêtre() {
        pane.add(new JLabel("Barre de status"), BorderLayout.SOUTH);
        pane.add(new JTree(), BorderLayout.WEST);
        textArea.setBackground(Color.GRAY);
        pane.add(textArea, BorderLayout.CENTER);
        JPanel toolbar = new JPanel(new FlowLayout());
        toolbar.add(new JButton("Open"));
        toolbar.add(new JButton("Save"));
        toolbar.add(new JButton("Cut"));
        toolbar.add(new JButton("Copy"));
        toolbar.add(new JButton("Paste"));
        pane.add(toolbar, BorderLayout.NORTH);    etc ..... }
}
```

// Sauver la fenêtre dans un fichier

```
public void saveWindow() throws IOException
{
    FileOutputStream fos = new FileOutputStream("window.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(window);
    oos.flush();
    oos.close();
}
```

// Reconstruire la fenêtre depuis fichier.

```
public void loadWindow() throws Exception
{
    FileInputStream fis = new FileInputStream("window.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    JFrame window = (JFrame)ois.readObject();
    ois.close();
    window.setVisible(true);
}
```